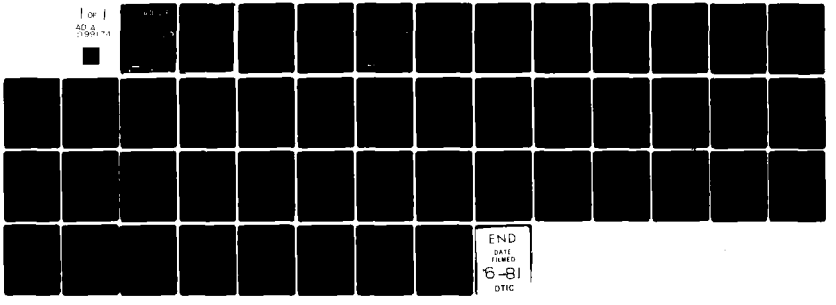
AD-A099 174 RAND CORP SANTA MONICA CA F/G 9/2
AN ANALYSIS OF PROXIMITY-DETECTION AND OTHER ALGORITHMS IN THE --ETC(U)
MAR 81 W S FAUGHT, P KLAHR F49620-77-C-0023
UNCLASSIFIED RAND/N-1587-AF NL



END
DATE FILMED
6-81
DTIC

LEVEL II

12

A RAND NOTE

AN ANALYSIS OF PROXIMITY-DETECTION AND OTHER ALGORITHMS IN THE ROSS SIMULATOR

William S. Faught, Philip Klahr

March 1981

N-1587-AF

Prepared For

The United States Air Force

Rand
SANTA MONICA, CA. 90406

The research reported here was sponsored by the Directorate of Operational Requirements, Deputy Chief of Staff/Research, Development, and Acquisition, Hq USAF, under Contract F49620-77-C-0023. The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon.

Published by The Rand Corporation

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: N-1587-AF	2. GOVT ACCESSION NO.: AD-A094
3. RECIPIENT'S CATALOG NUMBER:	
4. TITLE (and Subtitle): An Analysis of Proximity-Detection and other Algorithms in the ROSS Simulator	5. TYPE OF REPORT & PERIOD COVERED: Interim
	6. PERFORMING ORG. REPORT NUMBER:
7. AUTHOR(s): William S. Faught, Philip Klahr	8. CONTRACT OR GRANT NUMBER(s): F49620-77-C-0023
9. PERFORMING ORGANIZATION NAME AND ADDRESS: The Rand Corporation, 1700 Main Street, Santa Monica, CA. 90406	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 12
11. CONTROLLING OFFICE NAME AND ADDRESS: Requirements, Programs & Studies Group (AF/RDQM), Ofc, DCS/R&D and Acquisition, Hg USAF, Washington, DC 20330	12. REPORT DATE: March 1981
	13. NUMBER OF PAGES: 39
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public Release: Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

No Restrictions

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Algorithms
Computerized Simulation
Scenarios

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

See Reverse Side

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Summarizes the mechanisms by which the ROSS simulator computes interactions (collisions and proximities) between objects. ROSS simulates an air penetration scenario and is being developed to research techniques for improving large-scale simulation. The basic algorithm is analyzed in detail to determine its feasibility in the context of large numbers of objects, and to determine where improvements in speed can occur. (author)

A RAND NOTE

AN ANALYSIS OF PROXIMITY-DETECTION AND OTHER ALGORITHMS IN THE ROSS SIMULATOR

William S. Faught, Philip Klahr

March 1981

N-1587-AF

Prepared For

The United States Air Force

Rand
SANTA MONICA, CA. 90406

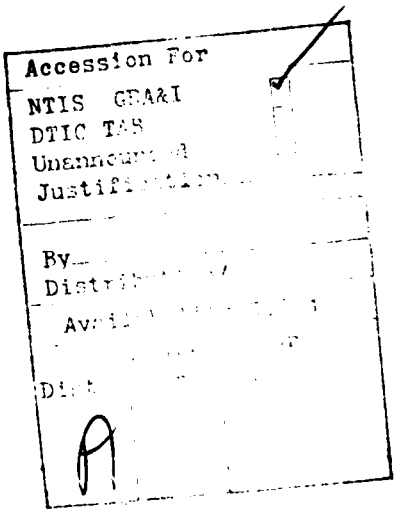
PREFACE

The ROSS simulator is a program that simulates a strategic air war scenario over enemy air space. It is being researched at Rand to test ideas that apply rule-based programming technology to simulation problems.

A key component of the air battle simulation is an algorithm, or procedure, that determines when objects are close enough to one another to cause some interaction. In large-scale simulations, this algorithm can be executed thousands of times.

This Note describes alternative algorithms for the job of "proximity detection," including the one developed for the ROSS program. Mathematical and experimental analyses are conducted to test the efficiency of the algorithm in different contexts.

The research described here is part of the project "Computer Technology for Real-Time Battle Simulation" supported by Project AIR FORCE to advance the technology and performance of large-scale simulators.



SUMMARY

The ROSS project has been investigating techniques to improve large-scale simulators. In particular, we are designing and implementing mechanisms to enhance their flexibility, understandability, and performance. Advances in Artificial Intelligence have provided tools (languages, man-machine environments, expert systems) which we have applied and extended to simulation. Our prototype simulator, called ROSS, currently runs a small-scale air penetration simulation. We are in the process of scaling up to more realistic levels.

The process of scaling up necessitates inclusion of more objects and more complex behaviors, which will result in increased computation time and slower simulations. We must make sure that the underlying algorithms are as efficient as possible. This Note examines these algorithms to determine their efficiency and whether they are suitable in larger contexts.

Within the domain of air penetration, a simulator must be able to detect collisions of objects as well as times at which objects can interact with one another (e.g., aircraft entering radar coverages). For efficiency, most air battle simulators compute times of future interaction initially and then schedule these potential interactions as events to be processed. During the simulation, the clock is moved to the time of the next scheduled event, and the event is processed. This approach (called "event-based") avoids computation during dead-time periods when nothing is occurring. This efficiency, however, is at the expense of flexibility, since objects are not allowed to modify their

predefined behavior (i.e., spontaneous or unexpected events cannot occur). Thus, for example, invading penetrators cannot modify pre-existing mission plans, and defensive fighters cannot be autonomously roving, looking for incoming penetrators.

To allow such flexibility, we have modified this approach to enable spontaneous actions and behaviors to occur while retaining the event-based scheduling mechanism. Our modified algorithm is explained in detail and is tested in various contexts. Through mathematical analysis and experimental testing, we found the algorithms to be computationally feasible for purposes of scaling up. Several suggestions for further improvements in efficiency are also presented.

CONTENTS

I. INTRODUCTION

The ROSS simulator [1,3,4] is a program which simulates a strategic air war scenario over enemy air space. Allied penetrators enter enemy air space, are potentially detected by radars (ground or airborne), and are intercepted by defensive fighters. ROSS is being built to test several ideas which apply rule-based programming technology to simulation problems. The hope is that this technology application will make large-scale simulators more tractable.

ROSS contains only a fragment of the complexity required for the eventual system. This Note addresses the question of how the simulator will behave as more objects are added to the simulation.

The purpose of this document is threefold:

1. To provide a brief overview of the basic algorithms used in the current version of the ROSS simulator.
2. To present an analysis of these algorithms, including an estimate of the resources required when the simulator is "scaled up" with more objects.
3. To present directly applicable suggestions for speeding up the algorithms based on the analysis.

One of the major design decisions to be made is the method of detecting collisions of objects or, more precisely, detecting when objects are close enough to one another to cause some interaction (e.g., within radar range). We will use the term "proximity detection" to refer to the determination of whether two objects can potentially

interact. The decision of how and when to perform such calculations has a major effect on the functionality and efficiency of the system. Large-scale battle simulators developed in the sixties chose an efficient but severely limited method of precomputing proximity detections before running the simulator. Because of these precomputations, these systems did not allow certain objects to modify their predefined behavior during the simulation. This limitation is not necessary and may result in unrealistic simulations (e.g., dynamic plan modifications may become the norm and not the exception). This document focuses primarily on the proximity detection algorithm and its parameters, with minor comments on other efficiency considerations.

This Note will not deal with analyses of partial, focussed, or less detailed simulation runs. The working hypothesis is that the entire simulation will be run on all objects.

II. OVERVIEW OF ROSS'S ALGORITHMS

As a first approximation, the ROSS simulator works like this: There is a set of objects, including penetrators (bombers), ground radars, fighters, fighter bases, filter centers (command and control centers), and targets. The penetrators move across the simulated territory until they enter the radar range of a ground-controlled intercept (GCI) radar. This detection then causes a chain of events to occur, e.g., the radar detects the penetrator and notifies its corresponding filter center (FC), the FC assigns and vectors a fighter to intercept the penetrator, and the fighter intercepts the penetrator.

Another event chain starts when the fighter intercepts the penetrator, tries to detect the penetrator, and subsequently fires missiles at it. A third event chain begins after a fighter intercepts a penetrator (or fails to); its FC can reassign the fighter to another penetrator, can ask the fighter to roam or loiter, or can send the fighter back to base. One additional penetrator detection can occur if a fighter has its radar on and is roaming; in this case, the fighter starts the intercept chain directly without informing the filter center. These event chains typically end when penetrators leave radar range and the fighter is sent back to base. Penetrators also have their own routing information that can be dynamically altered during the course of a simulation run, e.g., a penetrator can take an evasive turn after leaving GCI coverage and then reroute itself to its next target location.

Conceptually, then, all simulated actions in the current ROSS simulator are part of event chains. For the most part, ROSS initiates event chains when objects satisfy certain physical proximity conditions, the primary condition being the "proximity" of two objects. By proximity we mean two objects being within some interaction distance of each other, e.g., within radar range, within missile range, or actually colliding. Thus, all actions are the result of either:

(a) a proximity of two objects, or

(b) an action being caused directly by another action (as part of an event chain). This includes actions that were scheduled, by previous actions, to take place at certain specified future times (i.e., delayed to simulate real-time requirements).

For example, whenever a penetrator enters a GCI's radar range, ROSS starts a chain of events responding to the situation, e.g., the GCI detects the penetrator and notifies its FC for fighter assignment. An example of an action being scheduled involves the time required by GCIs to detect enemy aircraft: once a penetrator enters a GCI's range, the GCI notification is delayed to simulate the real time needed for detection.

The simulator itself is event-based. Events are either proximity checks or actions. Each event is a pair consisting of a simulation time (absolute time with respect to the start of the simulation) and an action. The event list is ordered by time. A simulation clock retrieves (and deletes) the next event to be run from the front of the event list, sets the clock time to that time, and then executes the

associated action. Actions may themselves add more events to the event list, again in order by time.

Note that the clock essentially skips all times from the current time to the next time that an event should run. The result is that there are no periodic or decaying functions that exist outside the clock-event-list mechanism. Periodic functions must schedule their periodicity by repeatedly putting events on the event list for their next execution. Conceptually, the designer must anticipate events occurring, such as collisions or fuel consumption. For example, to simulate fuel consumption, the simulator must check the aircraft's fuel when it first takes off, then schedule an event to check it later at some appropriate time. The later check must calculate the current availability, then reschedule a later event to recheck consumption.

We will divide the simulator algorithm descriptions into three categories: proximity detection, disengagement detection, and event chains.

2.1 PROXIMITY DETECTION

ROSS accomplishes proximity detection with a set of routines that monitor the interaction between every pair of significant objects (e.g., every penetrator with every working defensive radar). The algorithm is as follows: Two objects (as defined above) are tested to see if they are within a certain interaction distance (unique to the particular objects) of each other (e.g., a penetrator within a fighter's radar range). If they are, a chain of events begins, dependent upon the types of objects. If not, the objects are rechecked at a later time. The key

to the algorithm is that the objects are not rechecked until the earliest later time that they could interact, based on their locations and maximum velocities. The algorithm computes the time at which the two objects would be within interaction range of each other if they were headed directly toward each other at maximum velocity. The objects are checked for proximity again at that time. Thus, the algorithm is independent of any spontaneous or erratic behavior that either object may take that changes its velocity or direction.

2.2 DISENGAGEMENT DETECTION

In addition to calculating object engagements, ROSS must calculate one type of disengagement: penetrators exiting enemy radar ranges, e.g., GCI radars. This can be accomplished in one of two ways:

(1) with an algorithm similar to the one above: the simulator calculates the earliest time in which the penetrator could leave radar range, assuming it is headed at maximum velocity directly away from the center of the range, or

(2) by calculating the exact exit point, using an intercept formula, and requiring the simulator to detect any velocity changes the penetrator makes within radar coverage, and then recalculating the exit point.

Either of these two methods is viable. The current simulator uses the first algorithm. The second seems more attractive. Since the GCI must detect penetrator turns in any case, the same condition can be used for activating the exit point recalculation.

2.3 EVENT CHAINS

The major event chains and their triggering conditions are:

1. When a penetrator enters a GCI's range: the GCI detects the penetrator and notifies its corresponding filter center; the filter center then assigns and vectors fighters to intercept the penetrator.

(Note: The fighter's intercept, with its assigned penetrator, may or may not be calculated by the proximity detection algorithm above. The fighter's assignment is to proceed to a certain location and attempt to detect the penetrator. This scenario does not require the proximity detection algorithm, only a local proximity check when the fighter arrives at its intercept location. If, however, the fighter is allowed to detect any object in its path on the way to the intercept point, the proximity detection algorithm must be used.)

2. When the fighter arrives at the intercept point: the fighter attempts to detect the penetrator, fires missiles until it successfully kills the penetrator, and then notifies its filter center of its results.

3. When a fighter has completed intercepting a penetrator: the filter center either revectors the fighter to the same penetrator (if not killed), reassigns the fighter to another penetrator, lets the fighter roam to detect other penetrators, or sends the fighter back to its base.

4. When a penetrator leaves GCI coverage: the penetrator makes an evasive turn, flies the new course for some amount of time, then reroutes itself to its target.

These event chains also contain delays, corresponding to real-time requirements of their simulated actions. (As a sidelight, it is important to be aware of and resolve any conflicts involving the interactions of a delayed action and a proximity check, both of which may initiate event chains which may conflict with each other.)

III. ANALYSIS OF THE ALGORITHMS

In this section we analyze ROSS's algorithms to determine the effects of scaling up the number of objects. Various experiments have been run to examine the efficiency of these algorithms and determine points at which improvements can be made.

For the analysis, we considered two statistics:

1. the number of times a function is executed, and

2. the average execution time for one function execution.

We will focus on the number of executions of the major functions, leaving function execution times for later consideration.

A quick survey of the functions suggests that the analysis can be broken into three parts corresponding to the three categories discussed above: proximity detection, monitoring objects within radar range and detecting disengagements, and event chains. (We ran the simulator to count the number of executions for various functions and found that the distribution of executions indeed fell into the above three categories.)

3.1 EVENT CHAIN ANALYSIS

The third category, the analysis of event chains, is the easiest to dispose of. All the events in all the chains are primarily in one-to-one correspondence with the initial detection of a penetrator by a GCI. That is, each initial detection (of a continuous detection by GCIs) causes one event chain and culminates in an intercept of the penetrator by one or two fighters. These event chains are the heart of the

simulator. If the purpose of the simulator is to model events at the level of individual penetrators, GCIs, and fighters, and to individually assign and vector these objects, then the number of function executions corresponding to events in this third category cannot be reduced.

The one exception in this third category is the calculation of fighter-penetrator intercept locations and times. Every time that a filter center is notified of a penetrator detection, it calculates the possible intercept locations and times for each of its available fighters, selecting the closest to assign to the penetrator. Thus, the number of times the intercept function is called is O(D * F), where D is the number of detections and F is the average number of fighters available to be assigned. Some simple techniques (discussed later) can reduce the number of fighters that the filter center needs to consider for a given detection.

Thus, the major concern for the number of function executions focuses on proximity detection and monitoring for disengagements.

3.2 PROXIMITY DETECTION ANALYSIS

There are two straightforward methods for computing proximities with which we will contrast our algorithm:

(a) Update and examine the locations of objects every N ticks of a simulation clock, checking for proximities. (Note that for N=1, this is a time-stepped simulation which can be quite inefficient unless there is a high degree of interaction among objects at every tick.)

(b) Calculate the routes of all objects in advance and precompute

all potential object proximities; then recompute for every unanticipated change in velocity or direction during the simulation run.

Our algorithm is an adaptation of (a), and is obviously more efficient because it requires fewer proximity checks. For objects very far apart, our algorithm reduces the number of checks to O(1). For objects approaching each other closely, the algorithm approaches the behavior of (a).

Method (b) becomes quite inefficient if objects often make unanticipated changes in velocities and directions during a simulation run. For each change, the simulator would have to recompute future proximities of the changed object with all other objects. Thus, method (b) requires that the paths of all objects be known in advance, with few exceptions. For the current ROSS, this is not the case: any roaming fighter can detect penetrators if the fighter has its radar turned on. Such fighters can change their paths based on calculated intercepts with penetrator paths, which may, in turn, be influenced by any evasive maneuvers taken by the penetrators.

A modification to method (b) would be to calculate the potential proximities within a certain time frame. The proximity detection algorithm provides a rough approximation of this modification for objects far from each other. For close objects, and for short time frames in which neither object changes its velocity, this modification may be useful. It is listed in the suggestions section as a possible improvement to our algorithm.

The proximity detection algorithm is efficient for objects far from each other. The minimum proximity time for such objects will be large; thus, they will seldom be rechecked. For close objects, however, the unmodified algorithm can be quite inefficient. For example, two objects flying close to each other in parallel will be checked often.

To analyze the algorithm, we need estimates of the number of times the proximity-check test is made. Two objects are "checked" by testing to see if they are colliding or are in radar range of each other. Most of the analysis concerns the number of proximity checks for two objects approaching and receding from each other, as a function of their velocities, minimum distances, and radar ranges (e.g., a penetrator and a GCI, or a penetrator and a fighter). We then generalize over varying distributions of penetrators and radars.

In addition, we propose several modifications to the proximity detection algorithm, based on initial estimates of the above and analysis of the real events being simulated. We shall make similar estimates for the modified algorithm.

The earliest time that two objects could interact depends upon the distance between them and their maximum potential velocities. The actual time when they will interact depends upon the direction they travel and their actual speed. Thus, the number of times the objects must be checked in our algorithm depends upon how these two sets of factors compare with each other: If the actual velocity is much lower than the maximum possible, more checks will be needed than if the object flies at its maximum possible velocity. Similarly, if one object flies close to another but almost parallel to it, more checks will be needed

than if the two were on a direct collision course. We shall separate the two factors in our analysis.

3.2.1 Maximum vs. Actual Velocity

To isolate the speed component from direction, we shall consider the case of a penetrator P heading straight for the center of a GCI G. Let x be the initial distance of P from the closest point of G's radar, v be the velocity of P, and U be P's maximum possible velocity. We will calculate i, the number of checks made from the time the penetrator is at distance x to the time it enters the GCI's range.

If $v = U$, then P will enter G's range at the earliest possible time, and only 1 check is needed. If $v < U$, then an infinite number of checks will be needed before P enters G's range, because the earliest time P could enter G's range is always less than the time it will actually take. However, the time granularity of the simulator prevents this from happening. The simulator only schedules events to occur at whole seconds. (Fractions of seconds are rounded up to the next second.) Thus, when P is within one second's flying time from G, only one more check will be needed.

Figure 1 shows the situation when P is flying directly towards G.

Let

D_i = distance between P and G's radar after ith check.

Thus,

$$D_0 = x.$$

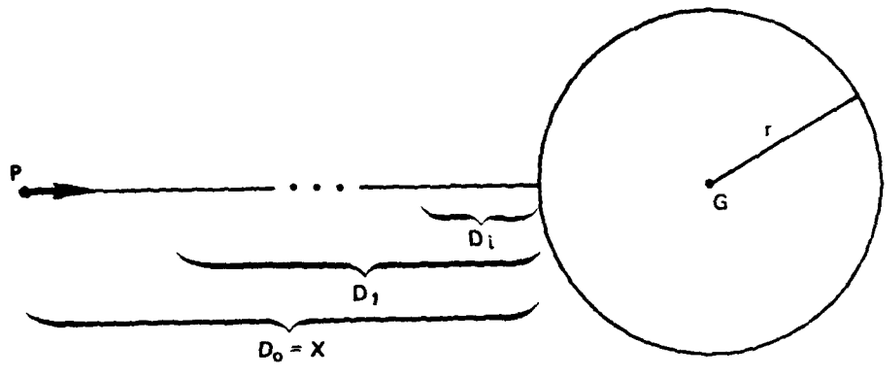


Fig. 1

The earliest P can enter G's range is

$$D_0 / U.$$

P will fly a distance of

$$(D_0 * v) / U$$

in that time. Thus,

$$D_1 = D_0 - (D_0 * v) / U$$

$$D_i = D_{i-1} - (D_{i-1} * v) / U$$

$$= D_{i-1} * (1 - v/U)$$

$$= D_0 * (1 - v/U)^i$$

$$i = \frac{\ln(D_i / D_0)}{\ln(1 - v/U)} .$$

Let m' be the minimum distance P can travel in one second. Assume

$$D_i = m'$$

i.e., at the next second, P will enter G's radar. Then the total number of proximity checks executed before P enters G's radar is

$$i = \frac{\ln(m'/x)}{\ln(1 - v/U)} .$$

As v -> 0, (1 - v/U) -> 1, ln(1 - v/U) -> 0, i -> infinity.

Figure 2 shows the situation when P is flying away from G.

Again, let

D_i = distance between P and G's radar after ith check,

and let

$$D_0 = m',$$

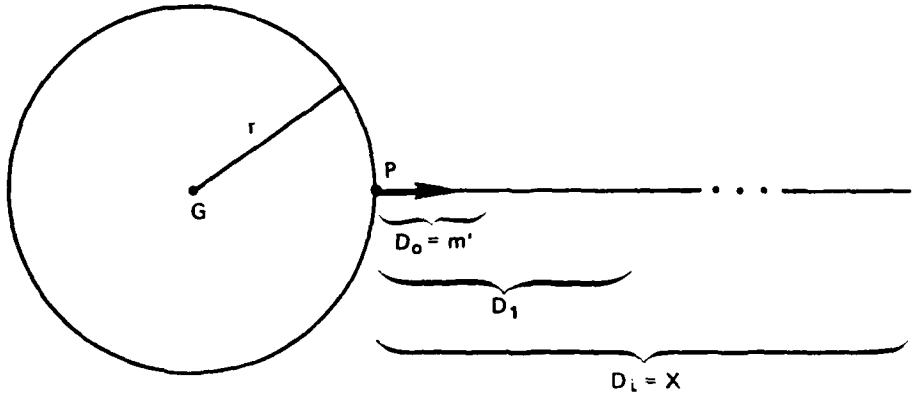


Fig. 2

the distance P flies in one second. The earliest P can reenter G's range is

$$D_0 / U.$$

P will fly a distance of

$$(D_0 * v) / U$$

in that time. Thus,

$$D_1 = D_0 + (D_0 * v) / U$$

$$D_i = D_{i-1} + (D_{i-1} * v) / U$$

$$= D_{i-1} * (1 + v/U)$$

$$= D_0 * (1 + v/U)^i$$

$$i = \frac{\ln(D_i / D_0)}{\ln(1 + v/U)} .$$

At a distance of x away from G's radar,

$$i = \frac{\ln(x/m')}{\ln(1 + v/U)} .$$

As v -> 0, (1 + v/U) -> 1, ln(1 + v/U) -> 0, i -> infinity.

Thus, as v becomes increasingly less than U, the number of checks grows rapidly. The obvious recommendation is to lower the maximum velocity to the lowest possible value, especially if the objects tend to move at a constant airspeed for the duration of the simulation.

For the remaining analysis (for direction of approach), we shall assume that U = v. Most of the estimators can be modified to account for U greater than v by replacing log base 2 by log base (1 + v/U), since i above can be rewritten as

$$i = \log_{(1 + v/U)} (D_i / D_0) .$$

3.2.2 Direction of Approach

We shall first consider the case of a penetrator approaching and either entering a GCI's range or bypassing it (without turning). (Later sections will consider distributions of penetrators and radars.)

A GCI G is assumed to be stationary and have a fixed circular radar range r. Assume penetrator P flies at a constant velocity (v = U) in a straight line path starting at some distance from the GCI and ending on the "opposite side" of the GCI an equal distance away. Let d be the (perpendicular) distance of the GCI's center to the penetrator's flight path.

There are five cases:

(1)	d >> r	P passes far away from G.
(2)	d > r, d close to r	P passes close to G.
(3)	d = r	P's path is tangent to G's range.
(4)	d < r, d close to r	P enters G's range.
(5)	d < r	P enters G's range.

To simplify the estimator, we will assume P starts at the closest point to G and flies away from G for some distance x; we will then double the value obtained for i.

Case 1: d >> r (Figure 3). The first proximity check is after P has flown D1; the second check is after P has flown D2; etc.

Since d - r is the shortest distance P could have flown to interact with G, then

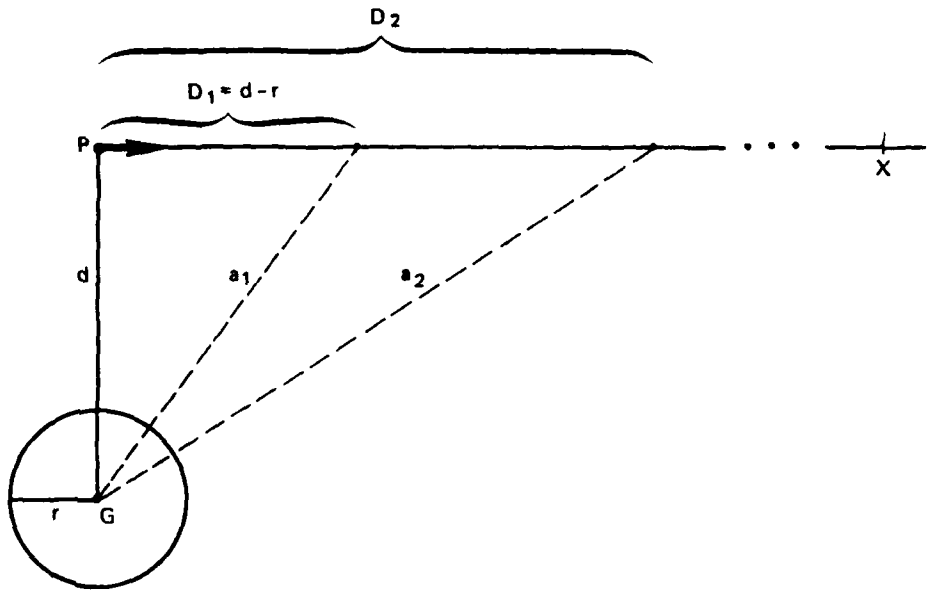


Fig. 3

$$D_1 = d - r.$$

Similarly,

$$D_2 = D_1 + a_1 - r$$

$$D_3 = D_2 + a_2 - r$$

$$D_i = D_{i-1} + a_{i-1} - r.$$

An upper bound on a_{i-1} is

$$a_{i-1} < d + D_{i-1}$$

by the triangle inequality. Therefore

$$a_{i-1} - r < d - r + D_{i-1}$$

which implies

$$D_i < D_{i-1} + d - r + D_{i-1}$$

$$D_i < 2\,D_{i-1} + d - r$$

$$D_i < 2^{i-1} D_1 + (2^{i-1} - 1)(d - r).$$

Since $D_1 = d - r$,

$$D_i < (2^i - 1)(d - r).$$

Let $x = D_i$, i.e.,

$$x < (2^i - 1)(d - r).$$

Because $d >> r$,

$$i \text{ approximates } \log_2 (x/d).$$

Case 2: d > r and d is close to r (Figure 4). In this case, r plays a large role. For example, if r is 1,000 and d is 1,001, P is

essentially flying parallel to G's radar range for a considerable distance. Call this distance z. Then for $x < z$, $x = i\,(d - r)$.

However, another factor comes into play. The simulator as a whole will have a minimum time granularity -- one second for the current ROSS. Assume the distance P can travel in the minimum time (one second) is m'. If $m' > (d - r)$, then $x = i\,m'$. That is, the minimum time until the next check is one second; therefore P will fly at least m' miles before the next check.

This suggests a modification to the algorithm to enhance its efficiency. Suppose the maximum time error for detecting proximities (call it M) were larger than one second. For example, if M = 5, then

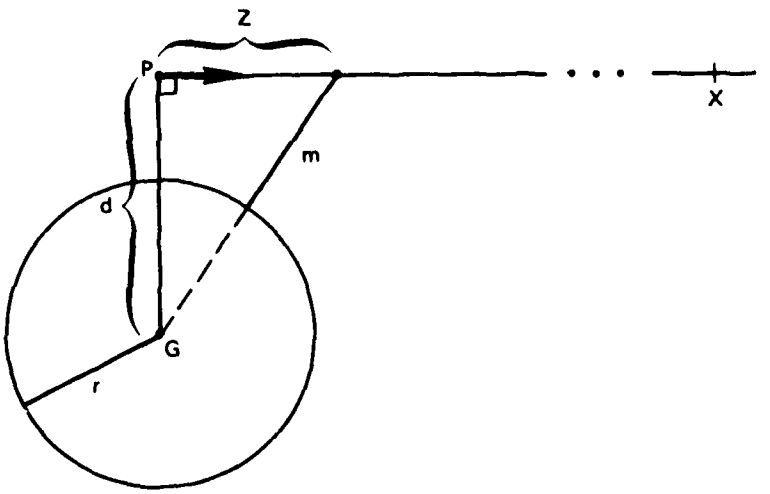


Fig. 4

all proximity detections will be accurate to within 5 seconds Let m be the distance P flies in M seconds. Then for d close to r, x = i m; and since m is larger than previously, i will be smaller. We shall assume this modification has been made for this case and the remaining cases.

The contributions to i will come from two sources:

(a) the checks when P is close to G's radar range; this part is dominated by m; and

(b) the checks when P is far away from G, in which i behaves more like case 1, because r + m > r. [Note: if r >> m this does not hold.]

The problem is to find z, estimate i for (a), and then use the case 1 formula for (b).

When P is close to G, the incremental distance between checks will be m. This will hold true until P is at least m away from G's range. We define z to be the distance P travels until P is a distance of m away from G's range. Thus, from Figure 4,

$$(m + r)^2 = z^2 + d^2$$

$$z = \sqrt{m^2 + 2mr + r^2 - d^2}\ .$$

For (a), the number of proximity checks until P travels z is z/m. For (b), we may use Case 1, subtracting those proximity checks that occur when P is close to G's radar. Thus, we have

$$i \text{ approximates } z/m + \log_2(x/d) - \log_2(z/d)$$

$$i \text{ approximates } z/m + \log_2(x/z).$$

[Again, if r >> m, this estimate will not work. Instead, z must be defined another way. This is not the case in the current ROSS, or in foreseeable applications, so we choose to ignore this case.]

Case 3: d = r, i.e., P's path is tangent to G's range. This is the most inefficient case for our algorithm, but it can be readily estimated as a special case of Case 2. Since d = r,

$$z = \sqrt{m^2 + 2mr}$$

$$i \text{ approximates } z/m + \log_2(x/z).$$

Case 4: d < r, and d close to r (Figure 5). Again, we define z as the distance P must fly until it is a distance of m away from G's range. In this case,

$$D_0 = z = \sqrt{(r+m)^2 - d^2}$$

$$D_1 = D_0 + a_0 - r = z + m$$

$$D_2 = D_1 + a_1 - r$$

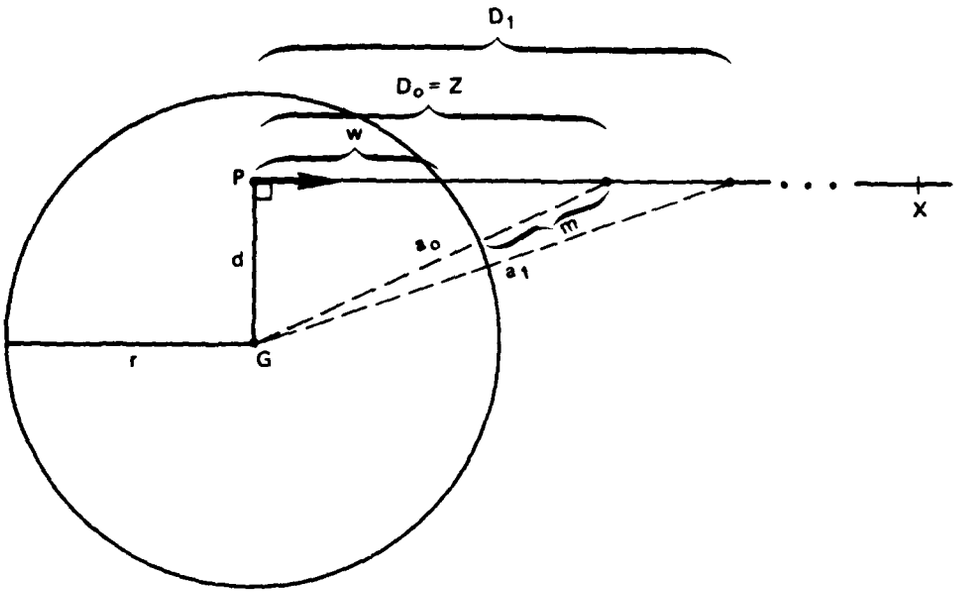


Fig. 5

$$D_{i+1} = D_i + a_i - r .$$

Since

$$a_i < D_i + d,$$

$$D_{i+1} < D_i + D_i + d - r$$

$$< 2 D_i + (d - r)$$

$$< 2^i D_1 + (2^i - 1)(d - r)$$

$$x < 2^{i}(z + m) + (2^{i} - 1)(d - r).$$

Thus, the number of proximity checks from the time P is at z to the time P is at x is

$$i > \log_2 \left(\frac{x + d - r}{z + m + d - r}\right).$$

To this we need to add the number of proximity checks from the time P leaves G's radar to the time P is at z. Thus,

$$i > \log_2 \left(\frac{x + d - r}{z + m + d - r}\right) + \frac{z - w}{m}$$

$$\text{where} \quad w = \sqrt{r^2 - d^2}\ .$$

Case 5: d < r, and d is small. In this case, as P flies away from G, Di doubles at each check. Thus,

$$D_{i+1} = 2^{i} D_0 = 2^{i} m$$

$$i = \log_2 (x/m).$$

When P flies towards G, however, the first check will be when P is very close to G. Thus, i is very small, and can be approximated by a constant.

The above analysis is for an engagement between a penetrator and a GCI. For fighters intercepting penetrators, there are two cases. The first is the strategy often used by classical simulators: The fighter would leave its radar off until the time it was scheduled to intercept the penetrator, then turn it on and try to detect the penetrator. If the fighter crossed paths with any penetrators on the way, the penetrators would not be detected. Thus, the proximity detection algorithm is not needed for this strategy.

The second case is that the fighter can detect penetrators that it comes across whenever it is in the air. In this case, the proximity detection algorithm must be started whenever a fighter becomes airborne. Two factors limit the number of checks that need to be made. The first is that the fighter is in the air for a limited amount of time compared with the time the simulator runs; the proximity detection algorithm can be turned off as soon as the fighter lands back at its base. The second is that the fighter will typically only interact with penetrators close to it because its assignments will be limited to penetrators in its own area.

3.2.3 Accuracy of Estimating Functions

We ran a number of test cases in the simulator and compared actual checks made with the estimated i. The results are given in Table 1. The estimating functions seem quite good except in the region from d = 1.01 r to d = 2 r. These functions require a better estimate of z, or perhaps a new third component of the equation for the section from the minimum distance covered to the logarithmically additive far section.

Recall that m is the minimum granularity for checks being made, i.e., m is the distance a penetrator could fly in M seconds, where M is the maximum time error for detecting proximities. Estimates close to the tangent case depended upon r not being much larger than m. In the current simulation, r is 25 miles, M is 5 seconds, and m is 1 mile. (That is, the error for detecting penetrators is 5 seconds.) Thus, m = .04 r, or a radar has a boundary that has a 4% error. We found that M = 5 was sufficient to significantly reduce i. When M = 10, little reduction was made over M = 5.

3.2.4 Distributions of Objects

Armed with an estimator function, we then calculated the number of checks for distributions of penetrators and GCIs. We simplified this part by assuming the following:

1. The number of checks for p penetrators and g GCIs is equal to p times the number of checks for one penetrator and g GCIs.

2. All distributions can be approximated by a single line of GCIs, with a penetrator intercepting the line at various angles. Pictorially, this can be represented on an x-y plane by a single vertical line of GCIs whose centers are on the y-axis, and a penetrator flying through the origin at various slopes.

3. If the penetrator intercepts more than 3 GCIs, it will be killed. Thus, the greatest angle is when the penetrator is tangent to the third GCI above the middle.

Table 1

Comparison of actual number of proximity checks to estimated number.

Initial and final distance from Penetrator to GCI: 250.0

m = 5, r = 25, d = closest distance between Penetrator and GCI

d	Actual Number	Estimated Number
0.1	9	10.2288187
5.0	9	10.2288187
10.0	11	10.2288187
15.0	11	11.1540442
20.0	13	11.9021556
22.5	14	13.2865173
24.0	16	15.5971725
24.5	19	17.4635124
24.8	20	19.7634819
24.9	21	21.2266185
24.99	21	24.183568
24.999	21	25.2712264
25.0	25	25.79977
25.001	25	25.79206
25.01	25	25.722475
25.1	24	25.0073988
25.2	23	24.166886
25.5	21	21.232878
26.0	18	15.9430871
27.5	14	13.3164229
30.0	11	11.3448507
35.0	9	9.4008794
40.0	8	8.285916
45.0	7	7.509775
50.0	6	6.91886324
60.0	6	6.0510703
70.0	5	5.4254361
80.0	5	4.94261146
90.0	4	4.5536804
100.0	4	4.23095447
150.0	3	3.169925
200.0	3	2.56021583
250.0	2	2.15600502

Let I be the sum of all of the i's for each penetrator-GCI pair (only one penetrator). We then calculated I (using the estimator functions above), varying the slopes of the penetrator, the uniform distances of the GCIs from each other, and the number of GCIs. The results are in Table 2. Multiplying by the number of penetrators gives the total number of checks for an entire simulator run, given in Table 3.

3.2.5 Execution Time Estimates

We also determined the execution time of each proximity check. In the current simulator, the time is 25 ms. If just the calculation is called, then the time is 2 ms. The extra time is consumed by the particular data base strategy used in Director [2] (the language upon which ROSS is implemented). This could be optimized so that the resulting check time would be 3 ms. Table 3 shows the time required for proximity detection for varying numbers of objects, assuming no other optimization or speedup techniques are used.

3.2.6 Comparison with Other Proximity Detection Models

As stated earlier, there are two other straightforward methods for computing proximities:

(a) Update and examine the locations of objects every N ticks of a simulation clock, checking for proximities.

Table 2

Calculation of the total number of proximity checks for one penetrator passing through a line of GCIs.

Initial and final distance from center Penetrator to GCI: 250.0

Distance between the centers of each neighboring pair of GCIs: 45 miles

m = 5, r = 25, slopes: 0, 1, and 6

Number of GCIs in the line	Total number of proximity checks
Slope = 0.0 :	
7	60
10	66
20	94
30	116
35	126
40	134
50	150
100	220
140	272
150	284
200	342
Slope = 1.0 :	
7	62
10	72
20	104
30	128
35	142
40	150
50	170
100	248
140	302
150	316
200	378
Slope = 6.0 :	
7	104
10	126
20	202
30	258
35	286
40	302
50	342
100	494
140	588
150	610
200	708

Table 3

Total number of proximity checks for all penetrators in a simulator run, using the results of Table 2.

Initial and final distance from center Penetrator to GCI: 250.0

Distance between the centers of each neighboring pair of GCIs: 45 miles

m = 5, r = 25, slope = 1

Each penetrator intercepts the equivalent of one line of GCIs at slope 1.

No. of Penetrators	10	50	250	1000
No. of GCIs	2	10	50	200
Proximity checks:	actual	actual	estimate	estimate
GCIs only	338	3234	42K	300K
Fighters and GCIs	583	7223	90K	600K
(of these, the number done to initialize ROSS)	(70)	(1750)	(12K)	(200K)
Times:				
At 25 ms/check: Fighters and GCIs	15 secs	180 secs	35 mins	4 hrs
At 2 ms/check: Fighters and GCIs	2 secs	15 secs	3 mins	20 mins

(b) Calculate the routes of all objects in advance and precompute all object proximities; then recompute for every unanticipated change in velocity or direction during the simulation run.

We found method (b) to be unacceptable because of its limited functionality: It is inefficient if we allow objects to exhibit spontaneous behaviors, e.g., spontaneous evasive penetrator maneuvers and spontaneous penetrator detections by fighters. Method (b) would be preferable only if the routes and velocities of the objects were known in advance. If this were the case, method (b) would be the most efficient algorithm for determining proximities.

Method (a) provides the same functionality as our algorithm, but at a much higher price. If the simulation is run with P penetrators and G GCIs for T time periods (in which the objects are checked for proximity once in each time period), then the total number of checks is

$$I = P * G * T.$$

This method assumes that the space of potential proximities is homogeneous, whereas in fact the space is non-homogeneous. Proximities only occur when penetrators enter a fixed, small (compared with the entire airspace path) region surrounding each GCI or set of GCIs. Our algorithm takes advantage of this fact by checking often when objects are close to each other, and less often when far apart.

The estimations take account for this non-homogeneity in two ways:

(1) The estimator function itself is logarithmic with respect to the distance between the objects.

(2) The distribution of GCIs with respect to penetrators assumes that additional GCIs will probably be distant from a single penetrator's path.

The total number of checks approximates

$$I = P * G * k$$

where k is the average number of GCIs that a penetrator will fly close to (i.e., within a distance of twice the GCI's range).

3.3 MONITORING ANALYSIS

When a penetrator enters a GCI's radar range, the GCI must monitor the penetrator to determine when the penetrator leaves its range. The current simulator uses an algorithm similar to the proximity detection algorithm. When the penetrator enters the GCI's range, the GCI calculates the earliest time at which the penetrator could leave the radar's range, i.e. assuming the penetrator was headed directly away from the GCI's center. The GCI then schedules another check at that time to recheck the penetrator's position. Thus, the algorithm is subject to the same potential inefficiency when the penetrator is heading a course that is almost tangent to the GCI's circular range.

We calculated the actual number of monitor checks needed for a penetrator intercepting a GCI at various distances from the GCI's center and found that the number was quite small. The maximum number needed was 15, given a minimum monitoring time of 5 seconds. Since the total

number of monitor checks for the entire run is bounded by the 15 times the number of times a GCI detected a penetrator, we did not pursue the analysis further.

IV. SUGGESTIONS FOR MODIFICATIONS

We have a number of suggestions for speeding up the algorithms based on the analysis. For the most part, these suggestions are directly applicable to the current design of ROSS.

In the event chains, the number of function executions for most functions are linear with the number of penetrator detections by GCIs. As stated above, these chains are an inherent part of the simulation and cannot be changed without altering the nature of the simulation.

However, the number of intercept calculations for determining when (or if) each fighter could intercept a particular penetrator is not linear: The calculation is done for each fighter whenever a penetrator is detected. The number of executions of this function could be reduced by only calculating intercepts for fighters within some maximum range, say 200 miles. (Calculating the distance between two objects is cheaper than an intercept calculation.) Also, for similar fighters (e.g., fighters with similar flight characteristics and payload) that are at the same fighter base, only one calculation need be done.

In the monitor calculation, the minimum monitoring time could be increased to reduce the number of proximity checks, but the time we used (5 seconds) already introduces an approximate 4% error in calculating the time at which the penetrator leaves the GCI's range. However, an entirely different algorithm may be useful: When the penetrator enters the GCI's range, the GCI calculates the exact exit point and time, and schedules another monitor check at that time (assuming no changes in the penetrator's velocity). If the penetrator ever changes direction or

speed, it must report the change (probably indirectly) to the GCI so the GCI can recalculate the exit point. It turns out that the penetrator must report velocity changes anyway, because the GCI must notice such changes and report them to the filter center. This algorithm seems straightforward to implement, although it only reduces the already small number of monitor function executions.

The number of proximity detection checks needed is a function of three factors, each somewhat amenable to modifications for improving performance:

1. The ratio of actual velocity to maximum possible velocity.

If the penetrators' actual velocities are constant for the entire simulator run, then their maximum possible velocities should be set to their actual velocities. Fighters will have more variable speeds, e.g., maximum for intercepting, minimum for loitering while waiting for assignment, and something between the two for patrols. Thus, the velocity ratio will have the most effect on fighter-penetrator engagements.

2. The angle of approach of the two objects.

As seen above, increasing the time granularity of the proximity detection algorithm can greatly improve the algorithm's performance for parallel or tangential approaches.

3. The distribution of GCI's.

A penetrator flying close to a block of GCI's, but entering none of

their ranges, will require multiple proximity detection checks, one set from each GCI. A possible modification to the algorithm is to consider a set of closely bunched GCI's to be one aggregate GCI with a range encompassing all of them. When the penetrator enters the aggregate range, the simulator breaks it apart (but only with respect to the one penetrator) and starts proximity detection checks between the penetrator and each GCI. Until that time, however, the simulator need only do proximity checks between the one aggregate GCI and the penetrator.

V. CONCLUSIONS

There are a number of tentative but well-supported conclusions that can be made from the analysis pertaining to scaling up the current prototype ROSS simulator:

1. The analysis shows proximity detection to be potentially the most time-consuming computational process in the simulator, but also clearly shows the algorithm presented here (with its modifications) to be computationally feasible.

2. The major portion of computational resources during a simulator run will probably be consumed by computing proximities; the major portion of development time for the simulator will be consumed in formulating and debugging the decision-based behavior of the simulator's objects.

Several other facts are indicated by our experience with developing the algorithm:

3. The scheduling of proximity-check events is potentially time- and space-consuming. A great deal of effort should be put into optimizing its time and space requirements. One option would be to enlist another processor just for scheduling proximity checks.

4. The proximity detection algorithm does not preclude dividing up the simulator's domain geographically and assigning a separate processor to compute all events for objects within its sector. But the algorithm will not benefit particularly from this division: Interactions between spatially distant objects consume few time resources. Such a division would, however, allow the processors to operate in parallel to enhance simulation speed.

REFERENCES

1. Faught, W. S., P. Klahr, and G. R. Martins. An artificial intelligence approach to large-scale simulation. Proceedings 1980 Summer Simulation Conference, Seattle, 1980.

2. Kahn, K. M. Director guide. AI Memo 482B, MIT, Cambridge, Mass., 1979.

3. Klahr, P., and W. S. Faught. Knowledge-based simulation. Proceedings First Annual National Conference on Artificial Intelligence. Stanford, 1980.

4. Klahr, P., W. S. Faught, and G. R. Martins. Rule-oriented simulation. Proceedings 1980 IEEE International Conference on Cybernetics and Society, Cambridge, Mass., 1980.


ATE
LME